File System Update

Possible approaches

Fri, May 7, 2004

Each front-end, whether an IRM based on the MVME-162 board or a PowerPC based on the MVME-2401 board, includes a nonvolatile memory-resident file system, which is used for keeping program and data files. The program files may be local applications, of which many may be active, or page applications, of which only one may be active at a time. A typical IRM uses 192K bytes for this memory, whereas a PowerPC node uses 1.5M bytes. When a program that is used in many nodes is updated, the copy stored in each node's file system must be updated. At present, this is done manually, although updating 160+ nodes can be tedious. This note explores how to do it automatically. In the end, a smarter manual update is preferred.

In principle, a manual update could be done using multicast targeting, so that all nodes would obtain the latest value at once. But there is a problem for IRMs, in that their limited available memory may not have enough contiguous space to receive a new updated copy of a program, which is often a bit larger than the previous version. In this case, the result is that the older version is deleted, but if that does not result in enough contiguous space available, the new one cannot be downloaded successfully.

Another problem is that multicasting reaches all nodes, whereas all nodes may not be running that program and therefore do not want it taking up space in their own file systems. But it is even worse than that. Because there are two types of nodes, the program files are different for each CPU. IRMs keep executable files; PowerPC nodes keep object files that are automatically linked during activation. In recognition of these two incompatible file versions, there are two "library" nodes. `Node0508` houses all IRM files, using a 1 MB nonvolatile memory card. `Node0619` houses all PowerPC files. For these reasons, a file is normally updated one node at a time. The Download page application, normally installed as Page D, makes this easy, as one only needs to change the target node number to move from one target to another. (It is neither necessary nor desirable to `telnet` into each node, for example.)

Automatic updating

Suppose we can assume that whatever file version is stored in the related library node is one that should be updated *in all nodes that need that file*. One could imagine still using multicast, but with some checks. Suppose one multicast address applied for all 68K nodes, and another applied for all PowerPC nodes. Also, if a node received a multicast message whose version date indicated it is newer than one already housed in that node, it might either go get it from the library node, or it could receive it as the rest of it is multicast. The assumption here is that one does not place a test update version on the library node, but only versions that are supposed to work.

An alternative scheme of *subscribing* to a library node for updates to specific files might be considered. When a new version is placed in the library node, it might target all the individual subscribers. This avoids the problems with multicasting in sending the code to nodes that do not need it. But why would not every IRM wish to subscribe to every file that is of common interest? If so, many transfers from the library would be required.

Advertising approach

The library node might periodically multicast the version dates. Each node can check whether the version date is newer, whether it has enough room, and arrange to download a copy to itself. Again, two multicast addresses would have to be assigned for this, to cover the two CPU types. This is a bit like advertising the availability of a software update. It is efficient in that it uses multicasting to advertise to all nodes. In the advertising message there might be included a minimum system version date, so that a system that is too old to run the new version would not pick it up.

One problem with the advertising approach, besides updating a program to an older system with which it is not compatible, is that a new local application may require different use of the parameter list, meaning that a new version cannot be loaded for execution until the parameter list has been suitably modified. Perhaps there is a way to download a new version but not automatically switch to it.

What communications mechanism should be used for this advertising purpose? If the feature is to be part of the system code, it could be done as a new listype. Or, since it is similar to one that already exists, that one may be modified to include the new support.

If the feature is to be supported via a local application, it could be done as a new Acnet protocol to which the new LA listens. It might be called `UPDF`, for “update file.” (The library node would likely run an advertising local application, say, called `ADVF`, for “advertise file.”) Every node that wishes to participate in the auto-update process would need to have this new LA enabled. This approach does not need to add to the multicast list at (`TRING+0x240`). No new UDP port must be defined, although there would be a new transient UDP port# used when the node accesses the library node to get a new copy of a given file. But this new feature is not Acnet-related.

How often must advertising take place? If a new node were built, how long should one wait before all relevant files are updated? This would not work to help populate a newly-created node, because only files already resident in a node are auto-updated. But suppose a new version is installed in a library node. How long is required before a node is updated with this version?

What size message can describe the current version date of a file? Consider a record of 16 bytes, where 8 bytes hold the file name and 8 bytes the version date, as 6 bytes of `YrMoDaHrMnSc` and two extra bytes. One of these can indicate the CPU type, which is (decimal) 40 for the MC68040 chip and 75 for the MPC750 PowerPC chip. Thus, an ethernet message of 1024 bytes, for example, could include 64 such advertising records. Library `node0508` currently houses 128 files, although it has space in its `CODES` table for up to 256 entries. Library `node0619` currently houses only 76 files, although it, too, can hold up to 256 entries.

A slow periodic multicast message can serve to keep things in synchronization well enough. But when a new version of a program file has been posted to a library node, it might be good to cause an immediate advertisement of that version, so that interested nodes can pick up the new version very soon.

An attitude about such changes that has been followed heretofore is that one should not make file changes unless it is needed, for fear that making any change might cause disruption in operations. ("If it ain't broke, don't fix it!") With that attitude, we can get into the situation where a number of files have been upgraded, but only a few nodes have been updated with those latest versions. It may be that we need some moderation in this advertisement/update approach, which assumes that any change made to a library node should be spread to the other nodes. If we only manually operated the advertisement step, we could wait to do it at a "safe" time. But what is safe for one node may not be safe for others. Of course, if a few nodes are doing more mission critical work, such that they should not even be in this scheme, they could simply not have their auto-updating LA enabled. That would mean they are in a situation the same as now; their programs would be updated only manually.

Let us assume that most nodes want the auto-updating feature, so that their auto-updating LA is enabled. Then, when one updates the version in a library node, one recognizes that this will likely update (nearly) all nodes that use that file. One would not likely want to update the library node without provoking an advertisement update. The chief advantage of the approach over simply multicasting the latest version is that only the nodes that need a given file will update it, not all nodes receiving the multicast advertisement message.

Manual approach, redesigned

What if we instead just implement a new listype just like the one used by Page D for copying files, except that it will not work unless several conditions are met:

1. The CPU type matches that of the local station
2. The program has a valid entry in the `CODES` table
3. The version date is different from the one in the `CODES` table
4. There is sufficient space to contain the new file version

Except for point #2, we could modify the logic of the present code that supports listype 76 to make these additional checks. But suppose we do modify the present support to include the above checks, but we condition the #2 check on whether the setting was multicast. If it is not multicast, then the program is meant to be copied. If it is multicast, it should only be copied if there is a `CODES` table entry for it already. One problem here is that a setting routine does not realize that a setting was sent via multicast; indeed, a setting routine is not formally related to the network at all.

For point #1, it is hard to decide whether we have the proper type of program. Maybe this can be left to the user, just as it is now. But if we target node `09F9`, we target all nodes. So how can PowerPC nodes determine which CPU type is being downloaded? Just because it came from an IRM, it may be that the IRM, while running Page D, got it from a PowerPC before passing it to the target PowerPC. So identifying the nature of the source node is not sufficient. If a program file is being transferred, which right now only includes file types `PAGE` and `LOOP`, an examination of the initial contents can do the trick. For PowerPC nodes, the first part of the object file is a `PEF` key. For IRMs, the first part of the executable is a 68K `LINK` instruction, which begins `0x4E56`. So the rule might be that only PowerPC nodes can house PowerPC program files, but any node can house a given data file.

To determine whether there is enough space available to contain a newer version of a file, which is often somewhat larger, is a bit tricky for IRMs. There may not be space at the moment. But if the current version of the file were deleted, there may be enough. So the logic should look for this possibility. If there is not enough space, downloading of the file should be disabled, since doing so would delete the old version while being unable to accept the new one. As a result, the node ends up with no version.

Reestablish objective

Let us review the initial objective. The scheme above does not actually perform automatic updating of software changes. But it allows targeting a multicast address to share a new version of a file with other nodes, avoiding doing so if it is unnecessary or difficult, but doing so when it can. Nothing is automatic; the user consciously targets the multicast address when necessary to perform a wide-ranging update.

The code for listype 76 in IRM currently does not check that the various pieces downloaded completely fill the file and leave no holes. This was somewhat intentional, but it was not done this way for the PowerPC. In the latter case, each data segment must immediately follow the last one. This requirement could be added to the IRM. The result is that no file can be written unless it is entirely written, from beginning to end.

Download sub-protocol

During a download operation, three kinds of transfers occur. The first transfer announces the size of the given named file to be transferred. This is followed by one or more data transfers, each of which indicates a starting offset, thus causing the `SIZE` field, which was set to zero by the initial transfer, to be advanced by the number of bytes of setting data. The final one is the termination call, when the version date and checksum are passed. The additional check would expect to see data transfers that are both in order and contiguous, and a `SIZE` field that matches the initial size of the file.

New listype

What we have here is a new listype that includes special checks to avoid accepting a downloaded file under certain conditions. The chief program that uses this listype is the download page application, which can be easily amended to use the new listype when it is targeting a multicast node number.

The current support for listype 76 can include some additional checks. Concerning the CPU type for files types `LOOP` and `PAGE`, we can check for the contents of the start of the file. This will have to be done in `NEXTDL` at the time that the first data segment is transferred. If the check fails, we can stop the download by calling `MLIBER` to free the allocated block and then clearing `PTRD`. Further data segment settings will fail, because there is no `PTRD`, as will a `TERMDL`. Besides the CPU check, we can check to be sure there are no holes in the transfer, by requiring that every data segment be contiguous and, when `TERMDL` is invoked, by checking that all data segments have been received. We can also check that there is sufficient space to accept a new version, as there is no desire when copying a file to intentionally free an old version without having something to replace it. When one does need to simply remove a file but not replace it, the way to do it is to do a `INITDL` with zero indicated `dataSize` as the 4-bytes of setting data.

So the two new checks that are to be included for the new listype only are:

> The file currently resides in the `CODES` table.
> The version date differs from that of the current file.

If the first of these checks fails, we reject the setting in `INITDL`. But if the dates do not match, we will not know this with the present support until `TERMDL`, which is too late to abort the process. It seems that the setting data passed for `INITDL` will also need to include the version date, following the 4-byte size. This may mean that the version date is passed twice, once following the file size for `INITDL` and secondly following the checksum for `TERMDL`. Eventually, it may be possible to remove the one for `TERMDL`, but if it were missing now, the current date would be used for a version date, which is not what we want.

Details

In order to make the extra check on whether the CPU type matches, the logic in `INITDL` must be made more tentative, deferring any real action on the file transfer until the first data transfer (`NEXTDL`) takes place. The size and version date in the `INITDL` setting data must be retained somewhere besides the `CODES` table entry of the current file that may be replaced. One way to do this is to allocate a transient `CODES` table entry for this purpose. (Bear in mind that it is possible for more than one file transfer to take place at the same time, although not for the same target file.) Since each setting is separate, the transient entry should have a unique name, which may be the one's complement of the file name being downloaded, or some other variation of the real file name. Once the first `NEXTDL` setting arrives, with the offset value of zero, the final determination can be made whether the download will actually be allowed to take place. It is only at this point that the previous version of the file is discarded. Then the previous `CODES` table entry can be reused, and the transient one discarded.

Review the fields of a 32-byte `CODES` table entry:

Field	*Size*	*Meaning*
`TYPE`	4	Type name
`NAME`	4	File name
`SIZE`	4	File size
`CKSM`	4	Checksum
`PTRD`	4	Download ptr
`PTRE`	4	Execution ptr
`DATE`	6	version date in BCD as `YrMoDaHrMnSc`
`CCNT`	2	Diagnostic call count

The file size and version date arriving with the `INITDL` data are kept in the transient entry. The type and file names are set to the one's complement of the type and file name that is to be downloaded.

After the first `NEXTDL` arrives, for which the offset is zero, use `FINDP` to find the transient entry and thus resurrect the file size and version date. Perform the required checks. If successful, then delete the original file and initialize the original entry for receiving the file transfer; then delete the transient entry. If the checks were unsuccessful, merely delete the transient entry. This seems complicated.

Better idea

To further ensure that a new program file is not accepted unless it has been completely received without holes, etc, consider allocating dynamic memory for the various blocks that arrive. Then, during `TERMDL`, it is clear whether the file transfer will be accepted. Only then should we delete the old version and write a new version — releasing all the temporary blocks, of course. Doing it this way would probably require some time-out to be imposed on the transfers, so that the temporary memory is released and the file transfer aborted, if too much time elapses without the `TERMDL` call. Since we want to permit multiple transfers to be possible at once, as long as they are not the same file, we need some organization for the context of such transfers.

Envision a linked list of allocated file header blocks, each of which pertains to an active file transfer and also includes a linked list of allocated data blocks. A new low memory ptr variable should be defined to be the head for a linked list of active file transfers. (Alternatively, some small amount of static memory could also be used for this, as there won't be very many file transfers active at once; a relatively small limit on the number of simultaneously active transfers could be imposed.)

Data structures

What should be in the allocated blocks for this support? For the file header block, we need the 8-character file name, the maximum size of the file, the total number of bytes accumulated so far, a pointer to the first data block, the time of the start of the transfer, and a pointer to the next such file transfer block. Each data block might hold the offset for the enclosed data block, the number of bytes enclosed, a ptr to the next data block, and the data itself. If an abort occurs, then the header block and all its related data blocks are freed. When the termination setting occurs, or if a time-out occurs, the header block and all its data blocks are freed. In order to impose a time-out, some code should be invoked at 15 Hz that visits each active header block.

If a data setting (`NEXTDL`) or termination setting (`TERMDL`) is received, and there is no active header block in the linked list, it must be ignored. Only an initial setting (`INITDL`) can install, or reactivate, a header block. It is possible to have an active header block (for which a time-out has not yet occurred) and receive an initial setting for that very file, in which case, the header block is reactivated. As a byproduct, this can permit simultaneous targeting of the same file—maybe.

```
* Downloaded file header block
*
MBLKSIZE    DS     1    ;Block size
NEXTFH      DS.L   1    ;ptr to next file header block
MBLKTYPE    DS     1    ;Block type#

FILENAME    DS.B   8    ;File type,name
```

```
FILESIZE     DS.L   1     ;File total size
TIMEOUTC     DS.L   1     ;Time-out counter
FILEOFFS     DS.L   1     ;Offset into file, so far
FILEDBN      DS.L   1     ;Count of data blocks

FIRSTDB      DS.L   1     ;ptr to first file data block, if any
LASTDB       DS.L   1     ;ptr to last file data block, if any

OWNERTID     DS.L   1     ;owner task Id, used for pSOS

* Downloaded file data block
*
MBLKSIZE     DS     1     ;Block size
NEXTDB       DS.L   1     ;ptr to next data block
MBLKTYPE     DS     1     ;Block type#

DATAOFFS     DS.L   1     ;Offset in file to this data
DATABLKN     DS     1     ;Data block#
DATASIZE     DS     1     ;Size of following data

DATAAREA     DS     n     ;Data bytes this block
```

Relevant routines called by ***`SETPROG`***

The file name argument points to 8 characters; the first 4 are the type name, and the last 4 are the file name of that type. The file name and file offset come from the 14-byte ident. The special flag is set for the new listype designed for multicast file settings. It is zero for the usual listype 76 case.

```
INITDL (filename, offset, setting_data, nBytes, special_flag)
```

Search header block linked list for match on `filename`.
If found, free all linked data blocks plus header block.
Abort transfer if special flag set and file has no `CODES` table entry.
Allocate and append new header block to linked list.
Initialize it with `filename`, total file size, set `fileOffs` to zero.
Also set `firstDB` and `lastDB` to `NULL` and initialize time-out counter.

```
NEXTDL (filename, offset, setting_data, nBytes, special_flag)
```

Search header block linked list for match on `filename`.
If not found, ignore and exit.
Find end of linked list of data blocks.
If `special_flag` set, and file type is `LOOP` or `PAGE`, check for correct CPU type. (If not correct CPU type, `DELETEHB`.)
Check that setting data with indicated offset fits at this point and is within stated file size. If not, `DELETEHB`.
Allocate data block for `setting_data` and initialize it, by setting current offset, data block count, and data size.
Copy `setting_data` into data block.
Append data block to end of linked list.

```
TERMDL (filename, offset, setting_data, nBytes, special_flag)

    Search header block linked list for match on filename.
    If not found, ignore and exit.
    Find end of linked list of data blocks.
    Check that total size matches offset + data size in this block.
    (If not, DELETEHB.)
    If special flag set, check that file has different version date.
    (If not, DELETEHB.)
    If special flag set, check that file name is in local CODES table.
    (If not, DELETEHB.)
    Check that there is sufficient room available for downloading
    this file into nonvolatile memory.
    (If not, DELETEHB.)
    Free current file of same name, if present.
    Allocate space for new version.
    Copy into new version.
    DELETEHB.
```

```
DELETEHB (fileHdrBlk)

    Scan linked list of data blocks, freeing each one.
    Remove file header block from linked list.
    Free header block.
```

Note that an abort that occurs in either the `INITDL` or the `NEXTDL` routines results in no action by subsequent `NEXTDL` or `TERMDL` invocations for that same file.

Diagnostics for SetFile

The following notes describe the diagnostics that were included in the `SetFile` implementation, which is the new version of `SetProg` that is still called `SetProg`, which is the set-type routine associated with the downloading file listypes.

New support for downloading files into the nonvolatile memory file system may benefit from the inclusion of diagnostics. This may be especially helpful during debugging. Consider the following diagnostic record structure:

Field	*Size*	*Meaning*
`fName`	8	4-char file type, 4-char file name
`fSize`	4	file size
`dBlks`	2	#data blocks
`fStatus`	2	status word
`dTime`	4	time to download in μs
`wTime`	4	time to write file in μs
`fDate`	8	time-of-day in BCD

Where can this 32-byte record be recorded? It might be written into a data stream, but every node will need this, so it may be easier to define a fixed location for it. An area of only 1K in size would allow for 32 such records, which may be more than enough.

If a file transfer is aborted, there should still be a record for diagnostic purposes. One approach that may serve is to expand the File Header block format to include a few more fields. When `DeleteHB` is called to delete all data blocks plus the file header block, it can record the appropriate diagnostic record. One new field is a status word. When an error condition occurs, the code can write into that field a value that will be copied into a diagnostic record just before the file header block is freed. Other fields will be the elapsed time fields. The date can also be kept in the file header block when it is created.

If one designs the circular buffer in the same format as a data stream queue, one could optionally define it in the `DSTRM` table and therefore access its contents as a data stream.

List the possible error conditions that are detected by the new code:

Error#	*Condition*
0	(no errors)
1	missing data block, blocks not consecutive
2	invalid CPU
3	data block outside file size range
4	failed to allocate data block
5	missing data at end of file
6	`CODES` table full
7	new `INITDL` replaces active transfer
8	file header block time-out
9	`FillFile` error
10	not enough nonvolatile memory to allocate file (pSOS only)

Error numbers can be stored into the new `fileStat` field in the file header block. It will be copied by the `DeleteHB` routine (via `SFileLog`) to build the diagnostic record.

With the addition of the new diagnostic fields, the new 64-byte File Header block structure is as follows:

Field	*Size*	*Meaning*
`mBlkSize`	2	block size
`nextFH`	4	ptr to next file header block, if any
`mBlkType`	2	block type#
`fileName`	8	file type, name
`fileSize`	4	file total size
`fileDBN`	2	count of file data blocks
`fileStat`	2	file error status

`fileOffs`	4	offset into file, so far
`timeOutC`	4	time-out counter
`firstDB`	4	ptr to first file data block, if any
`lastDB`	4	ptr to last file data block, if any
`ownerTId`	4	owner task id, used for pSOS only
`baseTime`	4	base time to measure elapsed time
`eTimTotl`	4	elapsed time from `INITDL` to `TERMDL`
`eTimWrit`	4	elapsed time to write file to memory
`fileTime`	8	time of day for file transfer

Post-implementation note

The above scheme was implemented as a part of the new changes for smarter downloading in the `SetProg` module for the PowerPC system. The low memory buffer assigned is at offset `0x5400`. This buffer allows for 31 file transfer diagnostic records. The pointer to the linked list of active file transfers is at low memory location `0x6FC`.

In summary, the new file system update scheme is a new manually-driven download approach that provides several new features. The idea is to support multicasting, via a separate listype, for updating files in many nodes at once, ensuring that only those nodes that use a given downloaded file will get updated, and that files are updated only if the version date differs from that of the current version. In addition, program files are only accepted by nodes of the appropriate CPU architecture. The new listype #97, to be used when multicasting file transfers, is easily supported, as its listype table entry is the same as that of listype 76 but for the last byte being set to 1 rather than 0.

Care is also taken in IRMs that any file downloading is ignored if there is insufficient room available to allocate the file. Also, file downloading data segments must be consecutive from beginning to end. No action is taken with the file system until all segments have been transferred so that everything is known. Only then is the new file written, and for local applications, automatic switching to the newly-downloaded version takes place. The new scheme, once it has been installed in all nodes, should greatly facilitate file updating across a large number of nodes.